CS377P Assignment 6

Eryn Li (jl58683) & Estevan Diaz (etd349)

1. Numerical Integration for Estimate of Pi

1.1. How Pi is estimated with limits

By iterating over a smaller range with the same number of data points, the answer gets closer to the actual integration value.

1.2. True-Sharing

Threads	Running Time	Speed Up
1	2,270,748,966	95.96%
2	7,699,830,144	28.30%
4	10,595,331,029	20.56%
8	11,947,626,110	18.23%

Value computed for pi with 8 threads: 3.14159264949813055168

1.3. Atomic Operation

Threads	Running Time	Speed Up
1	3,678,921,575	57.23%
2	5,640,740,721	38.63%
4	8,476,556,551	25.70%
8	10,737,045,404	20.29%

3.14159265123950026677

1.4. False Sharing

Threads	Running Time	Speed Up
1	2,198,487,611	99.12%
2	1,834,565,982	118.78%
4	1,045,689,313	208.39%
8	569,031,770	382.96%

3.14159265123950026677

1.5. No False-Sharing or True-Sharing

Threads	Running Time	Speed Up
1	2,193,989,149	99.33%
2	1,105,116,975	197.19%
4	566,499,756	384.68%
8	304,047,190	716.37%

3.14159265123950026677

1.6. Summary

With true-sharing, there is one global value, which all threads are contributing to. To ensure synchronization, a lock is placed around the read and update to the global sum. The result of this is slower code than a sequential limitation. This is because the overhead of the synchronization constructs and the poor parallelization of a single lock. With one lock, all the threads must wait on each other to contribute an update to the global value.

The atomic operations give similar results to true sharing because instead of the threads waiting on a lock, the threads wait in a while loop for the other threads to update the global value. This reduces parallel computation and reduces speed up.

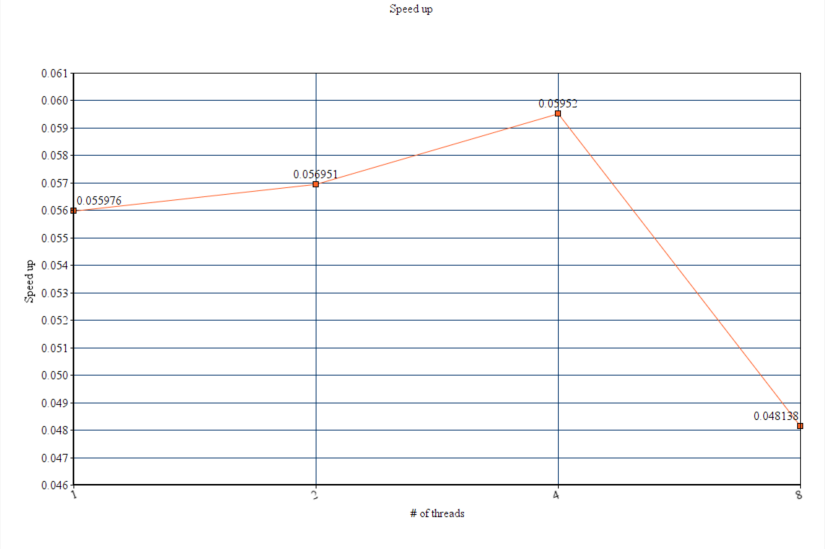
In false sharing, speed up is increased as the threads increase. However, the increase is not scaling with the number of threads. This is because a single array is being written to, therefore the threads are competing for the same slice of memory in their local caches, which they are all trying to write to. This reduces the amount of parallel computation and doesn't fully optimize our threads.

Without true sharing or false sharing, we see our speed up scale with the number of threads we use. This is because each thread will have a local, temporary, variable which they each individually update. The accumulated, partial, sum is not written until the thread is about to exit. This avoids competition with the cache and avoids threads waiting on each other.

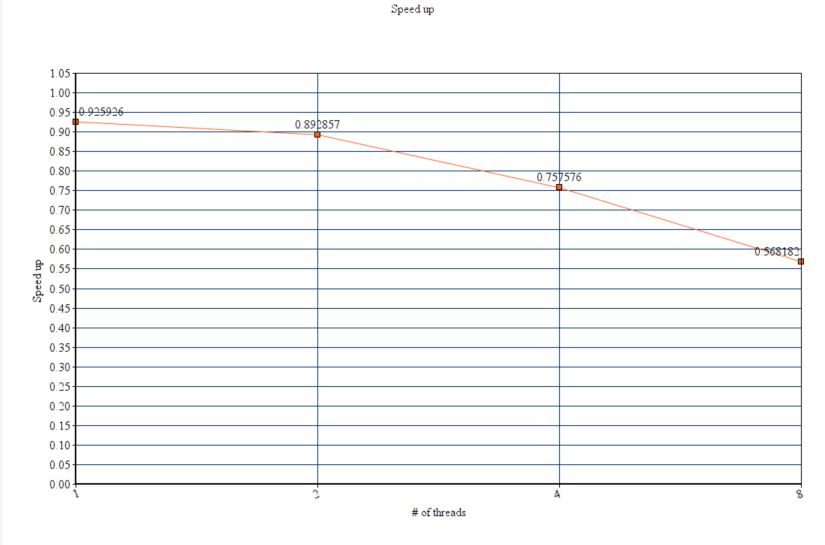
2. Parallel Bellman-Ford

2.1. Speedup with increasing threads

2.1.1. road NY



2.1.2. rmat15



2.2. Summary

We did not see speed up with the number of threads for either road networks or rmat graphs. This could have been for a number of reasons. Because we created new threads with each outer loop of the Bellman Ford algorithm, the overhead could have overshadowed the parallel computation benefits. Another reason might be poor cache performance due to all the threads accessing the same global array.